به نام خدا

گزارش تمرین ۳

محمد برآبادی ۹۷۲۳۰۱۵

۱.در ابتدا تمام اپراتورهای موردنظر را پیاده سازی کردم و این نکته مهم است که برای اپراتورها دو حالت وجود دارد که مقدار int و node کدام سمت چپ عملگر و کدام سمت راست آن است.

```
////////////////////////////////////////////////////////////////////////////////////////////////////
bool operator<(int _value,BST::Node node)
{
    if(_value < node.value)
        return true;
    return false;
}
////////////////////////////////////////////////////////////////////////////////////////////////////
bool operator<(BST::Node node,int _value)
{
    if(node.value < _value)
        return true;
    return false;
}
////////////////////////////////////////////////////////////////////////////////////////////////////
bool operator>(int _value,BST::Node node)
{
    if(_value > node.value)
        return true;
    return false;
}
////////////////////////////////////////////////////////////////////////////////////////////////////
bool operator>(BST::Node node,int _value)
{
    if(node.value > _value)
        return true;
    return false;
}
```

۲.یکی از مشکلاتی که من داشتم برای پیاده سازی تابع add_node بود که جای root عوض میشود و باعث میشود هر بار به جاهای مختلف اشاره کند و nodeها اشتباه ایجاد شوند.

```
////////////////////////////////////////////////////////////////////////////////////////////////////
bool BST :: add_node(int value)
{
    Node* root_1 {root};
    if(root == nullptr)
    {
        root = new Node{value, nullptr, nullptr};
    }
```

قبل از این جواب درست یک node درست میکردم و آدرس node را در root میریختم که همین باعث به وجود آمدن خطا میشد.

۳.تابع bfs را میتوان با stack ، vector یا queue نوشت که من از queue استفاده کردم و الگوریتم این تابع در اینترنت وجود دارد و به راحتی پیاده سازی کردم.

```
void BST :: bfs(std::function<void(BST::Node*& node)> func) const
{
    std::list<Node*> queue;
    queue.push_back(root);
    while(!queue.empty())
    {
        auto node =  queue.front();
        queue.pop_front();
        if(node !=  nullptr)
        {
            func(node);
            if(node->left)
                queue.push_back(node->left);
            if(node->right)
                queue.push_back(node->right);
        }
    }
}
```

۴.برای تابع length هم که از روش بازگشتی استفاده کردم ولی برای پیاده سازی رایج این تابع نیاز به یک تابع کمکی داشتم که بتوانم یک node به صورت ورودی به آن بدهم.

```
size_t bst_length(BST::Node *node)
{
    if (node == nullptr)
        return 0;
    else
        return(bst_length(node->left) + 1 + bst_length(node->right));
}
////////////////////////////////////////////////////////////////////////
size_t BST :: length()
{
    if (root->left == nullptr and root->right == nullptr)
        return 1;
    else
        return(1 + bst_length(root->left) + bst_length(root->right));
}
```

۵.تابع پرینت هم که به سادگی آدرس، مقدار،فرزند چپ و فرزند راست هر node را چاپ میکنم اما چون ورودی bst هست نیاز به یک تابع کمکی داشتم که بتوانم هر node را به آن بدهم و چاپ کنم.

```
std::ostream& operator<(std::ostream& os,BST::Node *node)
{
    os << node;
    os << "       => value:" << node->value;
    if(node->value < 10)
        os<< " ";
    os << "        left:" << node->left;
    if(node->left == nullptr)
        os<< "         ";
    os << "        right:" << node->right<<std::endl;
    if(node->left)
        std::cout < node->left;
    if(node->right)
        std::cout < node->right;
    return os;
}
////////////////////////////////////////////////////////////////////////////////////////////////////
std::ostream& operator<<(std::ostream& os,BST bst)
{
    os << "*********************************************************************************" <<
    os << bst.get_root();
    os << "       => value:" << bst.get_root()->value;
    os << "        left:" << bst.get_root()->left;
    os << "        right:" << bst.get_root()->right<<std::endl;
    if(bst.get_root()->left)
        std::cout < bst.get_root()->left;
    if(bst.get_root()->right)
        std::cout < bst.get_root()->right;
    os << "binary search tree size: " << bst.length() << std::endl;
    os << "*********************************************************************************" <<
    return os;
}
```

۶.در توابع find_node، find_parent و find _successor این نکته وجود دارد که خروجی double pointer میباشد و در طول پیاده سازی باید node* استفاده کنم.

در پیاده سازی هم از while استفاده کردم و تا زمانی که به جواب نرسیدم در درخت به سمت انتها حرکت کردم.

```
BST :: Node** BST :: find_parrent(int value)
{
    Node** root_1 {&root};
    while(1)
    {
        if((*root_1)->value > value)
        {
            if((*root_1)->left)
                if((*root_1)->left->value == value)
                    return root_1;
                else
                    root_1 = &(*root_1)->left;
        }
        else if((*root_1)->value < value)
        {
            if((*root_1)->right)
                if((*root_1)->right->value == value)
                    return root_1;
                else
                    root_1 = &(*root_1)->right;
        }
        else
            return nullptr;
    }
}
```

۷.یکی از چالش‌های مهم این تمرین، تابع delete_node میباشد که از تست ۱۶تا۲۳ برای این تابع میباشد.

از تابع find_node استفاده کردم تا بتوانم با استفاده از مقدار، node موردنظر را پیدا کنم و ابتدا بررسی میکنم که آیا این node وجود دارد یا نه؛ در ادامه هم که چهار حالت بررسی میشود که هیچ فرزندی نداشته باشد یا یک فرزند چپ یا راست داشته باشد یا دوتا فرزند داشته باشد.

به این باید دقت شود که این node که حذف میشود باید ارتباط پدر این node با فرزند(ها) این node برقرار شود تا درخت پیوسته بماند.

```
bool BST :: delete_node(int value)
{
    BST::Node** node{find_node(value)};
    if(node != nullptr)
    {
        if((*node)->left == nullptr and (*node)->right == nullptr)
        {
            BST::Node** node_par{find_parrent(value)};
            if(&(*node_par)->left == node)
                (*node_par) = new Node{(*node_par)->value, nullptr, (*node_par)->right};
            else
                (*node_par) = new Node{(*node_par)->value, (*node_par)->left, nullptr};
        }
        else if((*node)->right == nullptr)
        {
            BST::Node** node_par{find_parrent(value)};
            if(&(*node_par)->left == node)
                (*node_par) = new Node{(*node_par)->value, (*node)->left, (*node_par)->right};
            else
                (*node_par) = new Node{(*node_par)->value, (*node_par)->left, (*node)->left};
        }
        else if((*node)->left == nullptr)
        {
            BST::Node** node_par{find_parrent(value)};
            if(&(*node_par)->left == node)
                (*node_par) = new Node{(*node_par)->value, (*node)->right, (*node_par)->right};
            else
                (*node_par) = new Node{(*node_par)->value, (*node_par)->left, (*node)->right};
        }
```

```
        else
        {
            BST::Node** node_par{find_parrent(value)};
            BST::Node** node_succ{find_successor(value)};
            BST::Node** node_par_succ{find_parrent((*node_succ)->value)};
            if(&(*node_par_succ)->left == node_succ)
                (*node_par_succ) = new Node{(*node_par_succ)->value, nullptr, (*node_par_succ)->right};
            else
                (*node_par_succ) = new Node{(*node_par_succ)->value, (*node_par_succ)->left, nullptr};
            if(node_par != nullptr)
            {
                (*node_succ) = new Node{(*node_succ)->value, (*node)->left, (*node)->right};
                if(&(*node_par)->left == node)
                    (*node_par) = new Node{(*node_par)->value, (*node_succ), (*node_par)->right};
                else
                    (*node_par) = new Node{(*node_par)->value, (*node_par)->left, (*node_succ)};
            }
            else
                root = new Node{(*node_succ)->value, (*node)->left, (*node)->right};
        }
        return true;
    }
    else
        return false;
}
```

۸.چالش اصلی دیگر ما destructor و constructorها میباشند که خوشبختانه destructor به ما داده شده اما برای ۳تا constructor مخصوصا move و copy به اینترنت به افرادی که به این موضوعات مسلط هستند رجوع کردم.

```
BST::~BST()
{
    std::vector<Node*> nodes;
    bfs([&nodes](BST::Node*& node){nodes.push_back(node);});
    for(auto& node: nodes)
        delete node;
}
////////////////////////////////////////////////////////////////////////////////
BST::BST() : root{nullptr}
{

}
////////////////////////////////////////////////////////////////////////////////
BST::BST(BST &inpt_bst) : root{nullptr}
{
    inpt_bst.bfs([this](BST::Node *&node){this->add_node(node->value);});
}
////////////////////////////////////////////////////////////////////////////////
BST::BST(BST &&inpt_bst)
{
    root = inpt_bst.root;
    inpt_bst.root = nullptr;
}
```

۹. برای اپراتور = هم باید دو حالت copy و move را مینوشتم که از constructor آن‌ها استفاده کردم.

```
BST BST::operator=(BST &inpt_bst)
{
    inpt_bst.bfs([this](BST::Node *&node){this->add_node(node->value);});
    return *this;
}
////////////////////////////////////////////////////////////////////////////////
BST BST::operator=(BST &&inpt_bst)
{
    root = inpt_bst.root;
    inpt_bst.root = nullptr;
    return *this;
}
////////////////////////////////////////////////////////////////////////////////
```

۱۰.حالتی که به جای add_node یکسری عدد به صورت vector وارد میشوند از initializer_list استفاده میشود و باید تمام اعداد در این لیست را با استفاده از تابع add_node به درخت اضافه کنیم.

فقط برای این که به همه المان‌های لیست دسترسی داشته باشیم جستوجو کردم که از یک for استفاده کردم.

```
BST::BST(std::initializer_list<int> inpt) : root{nullptr}
{
    for(int i : inpt)
        add_node(i);
}
```

۱۱.برای تابع ++ باید به این دقت شود که در یکی مقدار فرستاده میشود سپس اضافه میشود اما در دیگری مقدار زیاد میشود سپس فرستاده میشود که از تابع bfs که در قسمت‌های قبل پیاده سازی کردم استفاده کردم.این قسمت نیازمند سرچ زیاد و حتی دیدن دوباره کلاس استاد بود.

```
BST BST::operator++(int inpt)
{
    auto bfs {*this};
    this->bfs([this](BST::Node *&node){(node->value++);});
    return bfs;
}
////////////////////////////////////////////////////////////
BST BST::operator++()
{
    this->bfs([this](BST::Node *&node){(node->value++);});
    return *this;
}
```

https://github.com/MBW0lf/AP_HW3

با تشکر